CLOUD APPLICATION DEVELOPMENT

(HDWD_SEP23OL)

Author: Luis Martins - x23194456

TABLE OF CONTENTS

Designing the Applications

PLANNING:

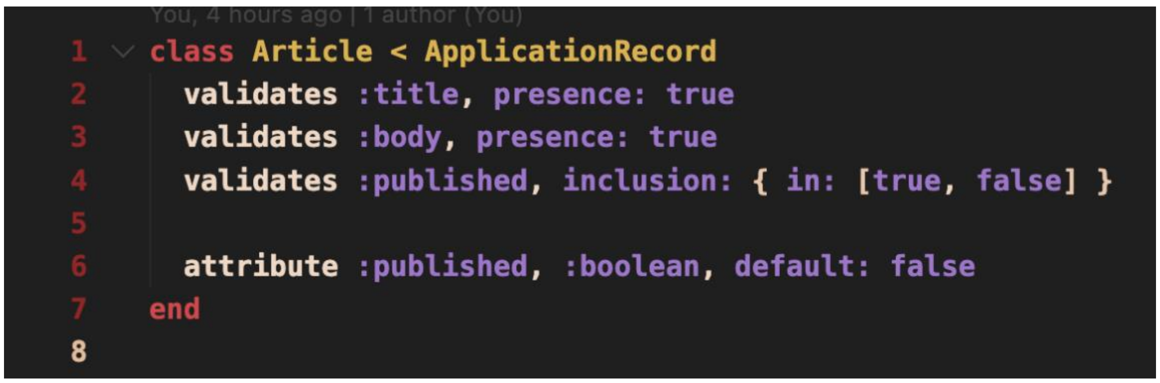
Analyse Requirements: I carefully reviewed the provided problem statement to understand the requirements for the full-stack web application.

Identify Features: I identified key features such as CRUD operations for articles, filtering articles, and implementing both backend and frontend components.

Architecture Planning: I determined the architecture of the applications, including the technology stack (Ruby on Rails, React, HTML), database design, and API design.

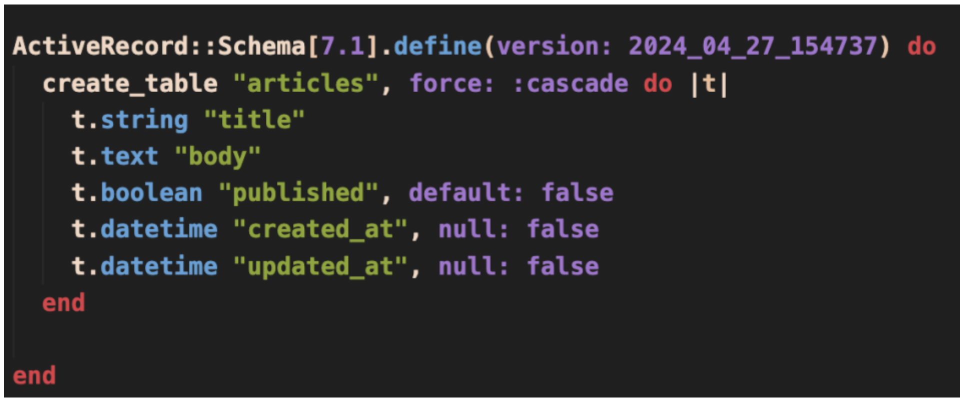
BACKEND APPLICATION (RUBY ON RAILS):

Model Design: I designed the **Article** model with attributes: **title** (string), **body** (text), and **published** (boolean, default: false).


You, 4 hours ago | 1 author (You)

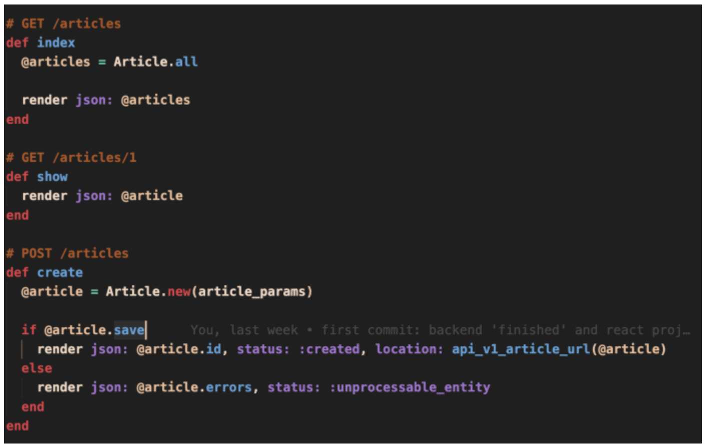
```ruby
class Article < ApplicationRecord
  validates :title, presence: true
  validates :body, presence: true
  validates :published, inclusion: { in: [true, false] }

  attribute :published, :boolean, default: false
end
```

Migration and Validation: I created a migration file for the **Article** model to set up the database table and added validations to ensure data integrity.



```
ActiveRecord::Schema[7.1].define(version: 2024_04_27_154737) do
  create_table "articles", force: :cascade do |t|
    t.string "title"
    t.text "body"
    t.boolean "published", default: false
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
  end

end
```

Controller Implementation: I implemented controller actions for CRUD operations on articles, including GET all articles, GET by ID, POST, PUT, and DELETE.

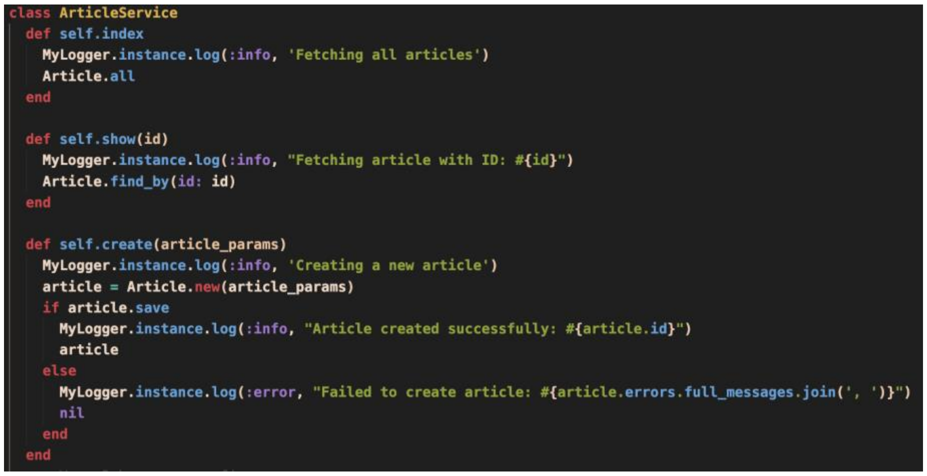


```
# GET /articles
def index
  @articles = Article.all

  render json: @articles
end

# GET /articles/1
def show
  render json: @article
end

# POST /articles
def create
  @article = Article.new(article_params)

  if @article.save      You, last week • first commit: backend 'finished' and react proj…
    render json: @article.id, status: :created, location: api_v1_article_url(@article)
  else
    render json: @article.errors, status: :unprocessable_entity
  end
end
```

Design Patterns: In designing the full-stack web application for article management, I employed various design patterns to enhance maintainability, scalability, and readability of the codebase. Specifically, I utilized the Service pattern and the Singleton pattern.

Service Pattern:

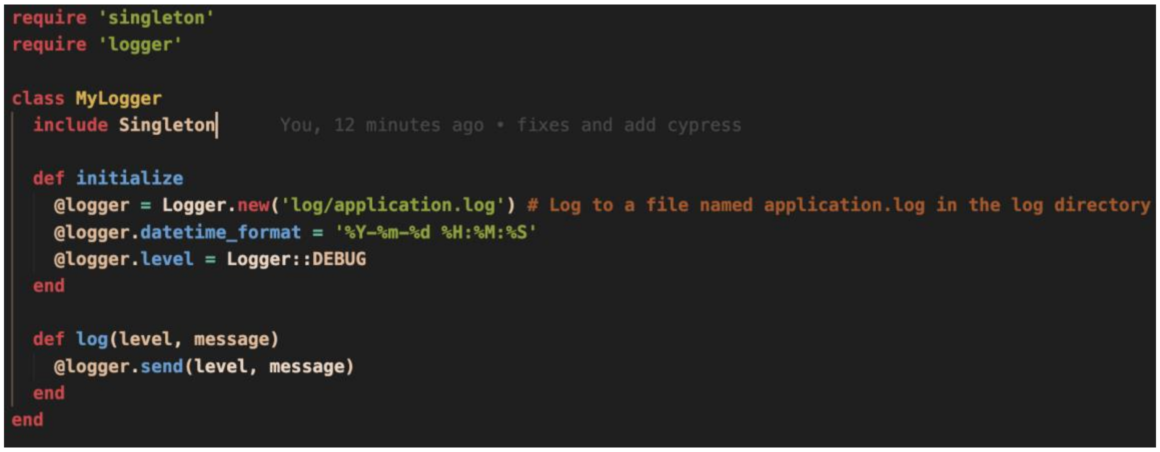
The Service pattern was implemented to encapsulate business logic and facilitate separation of concerns. I created a service class called **ArticleService** to handle CRUD operations on articles within the backend Rails application. This class abstracts the database interactions and validation logic away from the controller, promoting a cleaner and more modular codebase. Each method in the **ArticleService** class corresponds to a specific action, such as creating, updating, retrieving, and deleting articles. This pattern helps in maintaining a clear separation of concerns and promotes code reusability.



```ruby
class ArticleService
  def self.index
    MyLogger.instance.log(:info, 'Fetching all articles')
    Article.all
  end

  def self.show(id)
    MyLogger.instance.log(:info, "Fetching article with ID: #{id}")
    Article.find_by(id: id)
  end

  def self.create(article_params)
    MyLogger.instance.log(:info, 'Creating a new article')
    article = Article.new(article_params)
    if article.save
      MyLogger.instance.log(:info, "Article created successfully: #{article.id}")
      article
    else
      MyLogger.instance.log(:error, "Failed to create article: #{article.errors.full_messages.join(', ')}")
      nil
    end
  end
```

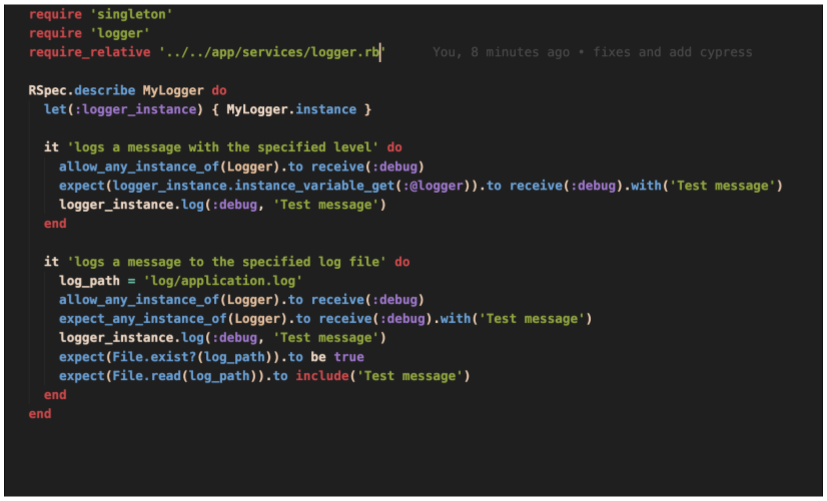
Singleton Pattern:

The Singleton pattern was employed to ensure that there is only one instance of the **MyLogger** class throughout the application's lifecycle. This logger class provides logging functionality to various components within the application. By using the Singleton pattern, we ensure that all parts of the application share the same logger instance, preventing unnecessary resource allocation and ensuring consistent logging behavior.



```ruby
require 'singleton'
require 'logger'

class MyLogger
  include Singleton      You, 12 minutes ago • fixes and add cypress

  def initialize
    @logger = Logger.new('log/application.log') # Log to a file named application.log in the log directory
    @logger.datetime_format = '%Y-%m-%d %H:%M:%S'
    @logger.level = Logger::DEBUG
  end

  def log(level, message)
    @logger.send(level, message)
  end
end
```

Testing: I wrote unit tests for the **Article** model and controller actions using **RSpec**.



```ruby
require 'singleton'
require 'logger'
require_relative '../../app/services/logger.rb'      You, 8 minutes ago • fixes and add cypress

RSpec.describe MyLogger do
  let(:logger_instance) { MyLogger.instance }

  it 'logs a message with the specified level' do
    allow_any_instance_of(Logger).to receive(:debug)
    expect(logger_instance.instance_variable_get(:@logger)).to receive(:debug).with('Test message')
    logger_instance.log(:debug, 'Test message')
  end

  it 'logs a message to the specified log file' do
    log_path = 'log/application.log'
    allow_any_instance_of(Logger).to receive(:debug)
    expect_any_instance_of(Logger).to receive(:debug).with('Test message')
    logger_instance.log(:debug, 'Test message')
    expect(File.exist?(log_path)).to be true
    expect(File.read(log_path)).to include('Test message')
  end
end
```

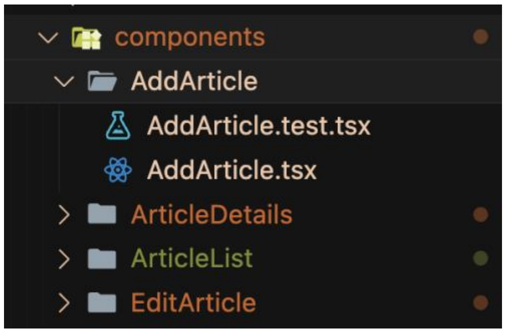
Frontend Application (React):

Component Development: I developed React components for various functionalities, including:

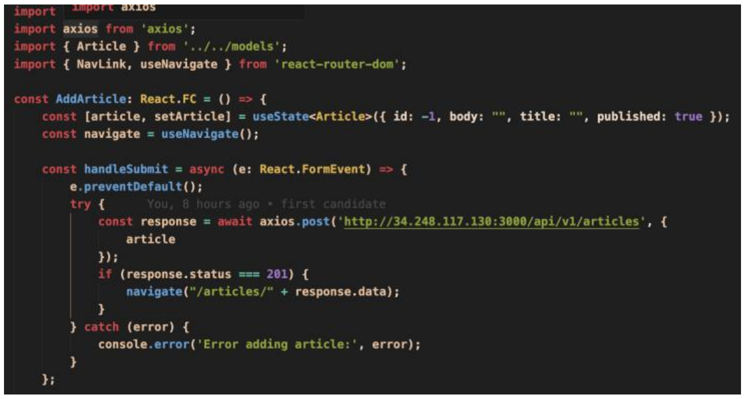
Displaying Articles: I created a component to render a list of articles fetched from the backend, displaying their titles, bodies, and publication status.

Adding/Editing Articles: I implemented forms for adding new articles and editing existing ones, allowing users to input titles, bodies, and publication status.

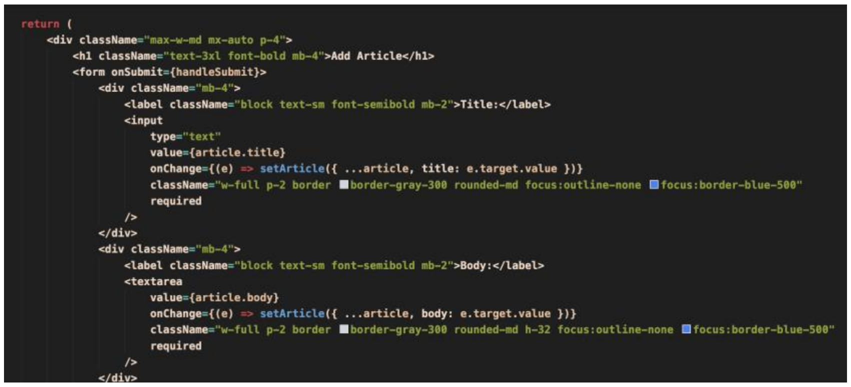
Filtering Articles: I designed a feature to filter articles based on their publication status, enabling users to toggle between viewing published and unpublished articles.



```
components
  AddArticle
    AddArticle.test.tsx
    AddArticle.tsx
  ArticleDetails
  ArticleList
  EditArticle
```



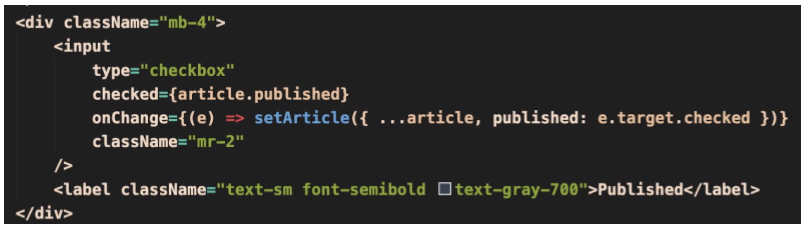
```
import   import axios
import axios from 'axios';
import { Article } from '../../models';
import { NavLink, useNavigate } from 'react-router-dom';

const AddArticle: React.FC = () => {
    const [article, setArticle] = useState<Article>({ id: -1, body: "", title: "", published: true });
    const navigate = useNavigate();

    const handleSubmit = async (e: React.FormEvent) => {
        e.preventDefault();
        try {       You, 8 hours ago • first candidate
            const response = await axios.post('http://34.248.117.130:3000/api/v1/articles', {
                article
            });
            if (response.status === 201) {
                navigate("/articles/" + response.data);
            }
        } catch (error) {
            console.error('Error adding article:', error);
        }
    };
```



```
return (
    <div className="max-w-md mx-auto p-4">
        <h1 className="text-3xl font-bold mb-4">Add Article</h1>
        <form onSubmit={handleSubmit}>
            <div className="mb-4">
                <label className="block text-sm font-semibold mb-2">Title:</label>
                <input
                    type="text"
                    value={article.title}
                    onChange={(e) => setArticle({ ...article, title: e.target.value })}
                    className="w-full p-2 border border-gray-300 rounded-md focus:outline-none focus:border-blue-500"
                    required
                />
            </div>
            <div className="mb-4">
                <label className="block text-sm font-semibold mb-2">Body:</label>
                <textarea
                    value={article.body}
                    onChange={(e) => setArticle({ ...article, body: e.target.value })}
                    className="w-full p-2 border border-gray-300 rounded-md h-32 focus:outline-none focus:border-blue-500"
                    required
                />
            </div>
```

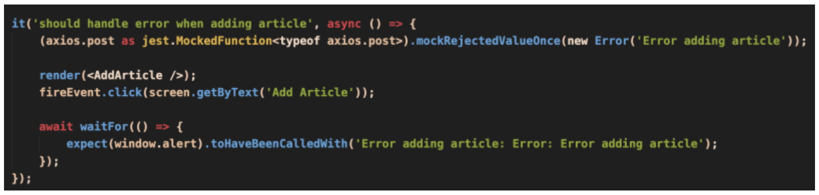
API Integration: I integrated the React client with the Ruby on Rails backend by making HTTP requests to the defined API endpoints.

Styling and UI Design: I designed the UI components using JSX and Tailwind, creating a user-friendly interface.

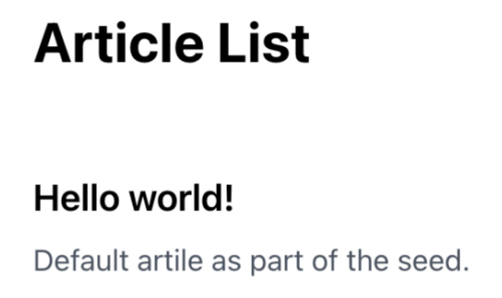


```
<div className="mb-4">
    <input
        type="checkbox"
        checked={article.published}
        onChange={(e) => setArticle({ ...article, published: e.target.checked })}
        className="mr-2"
    />
    <label className="text-sm font-semibold text-gray-700">Published</label>
</div>
```

Testing: I wrote unit tests for React components using **Jest** and **React Testing Library**.



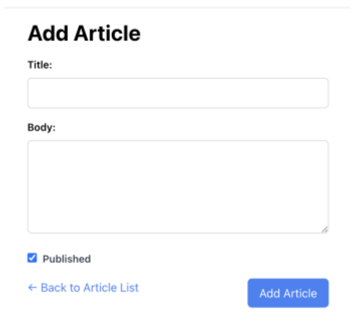
```
it('should handle error when adding article', async () => {
    (axios.post as jest.MockedFunction<typeof axios.post>).mockRejectedValueOnce(new Error('Error adding article'));

    render(<AddArticle />);
    fireEvent.click(screen.getByText('Add Article'));

    await waitFor(() => {
        expect(window.alert).toHaveBeenCalledWith('Error adding article: Error: Error adding article');
    });
});
```



Article List

Hello world!

Default artile as part of the seed.

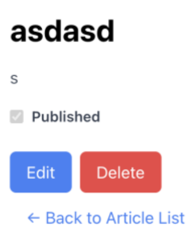


Add Article

Title:

Body:

☑ Published



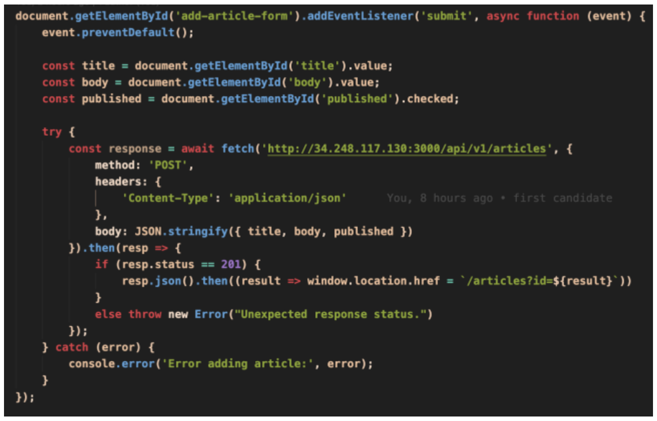
asdasd

s

☑ Published

HTML CLIENT:

Form Development: I designed intuitive forms and user interface elements using HTML and JavaScript to facilitate Create, Read, Update, and Delete (CRUD) operations on articles. By using HTML's structural elements and JavaScript's dynamic capabilities, I built an interactive and user-friendly experience for managing articles within the application. This involved designing forms for inputting article data and implementing responsive UI components to enable easy interaction with the article management functionalities. Through careful attention to detail and adherence to best practices in web development, I ensured that the HTML and JavaScript components effectively met the requirements of the application while providing a smooth and intuitive user experience.

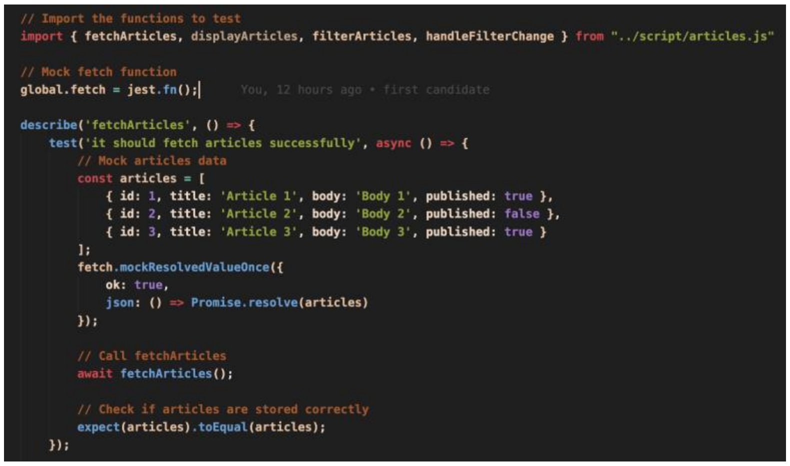


```
document.getElementById('add-article-form').addEventListener('submit', async function (event) {
    event.preventDefault();

    const title = document.getElementById('title').value;
    const body = document.getElementById('body').value;
    const published = document.getElementById('published').checked;

    try {
        const response = await fetch('http://34.248.117.130:3000/api/v1/articles', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'      You, 8 hours ago • first candidate
            },
            body: JSON.stringify({ title, body, published })
        }).then(resp => {
            if (resp.status == 201) {
                resp.json().then((result => window.location.href = `/articles?id=${result}`))
            }
            else throw new Error("Unexpected response status.")
        });
    } catch (error) {
        console.error('Error adding article:', error);
    }
});
```

HTTP Requests: I ensured that the HTML client makes appropriate HTTP requests to the backend API endpoints for each action.

Styling: I applied CSS styles to the HTML elements to enhance the visual presentation and usability.

Testing: I wrote tests to validate the HTML and JavaScript functionality using **Jest** again. These tests encompassed a wide range of scenarios, including user interactions, data validation, and event handling, ensuring that the frontend components functioned as intended across various use cases. By rigorously testing the HTML and JavaScript codebase, I aimed to identify and address any potential bugs or issues, thereby improving the reliability and stability of the application.



```
// Import the functions to test
import { fetchArticles, displayArticles, filterArticles, handleFilterChange } from "../script/articles.js"

// Mock fetch function
global.fetch = jest.fn();      You, 12 hours ago • first candidate

describe('fetchArticles', () => {
    test('it should fetch articles successfully', async () => {
        // Mock articles data
        const articles = [
            { id: 1, title: 'Article 1', body: 'Body 1', published: true },
            { id: 2, title: 'Article 2', body: 'Body 2', published: false },
            { id: 3, title: 'Article 3', body: 'Body 3', published: true }
        ];
        fetch.mockResolvedValueOnce({
            ok: true,
            json: () => Promise.resolve(articles)
        });

        // Call fetchArticles
        await fetchArticles();

        // Check if articles are stored correctly
        expect(articles).toEqual(articles);
    });
```

Note: both the React app and HTML client look exactly the same.

Testing and Integration

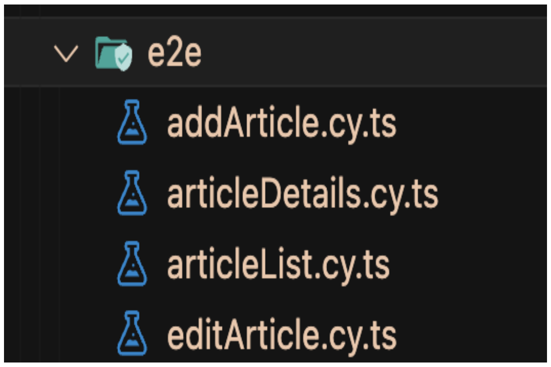
Integration Testing: I implemented integration tests to cover end-to-end scenarios for article creation, retrieval, updating, and deletion using **Cypress and Ruby test**. These tests encompassed every aspect of the application's functionality, including:

Article Creation: Ensuring that articles could be successfully created and added to the database.

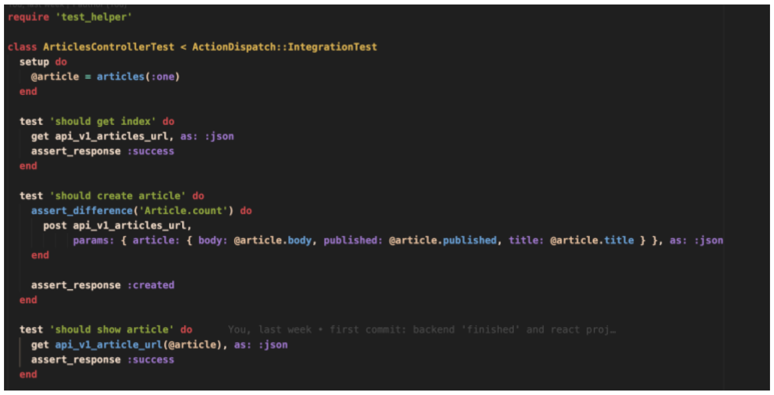
Article Retrieval: Verifying that articles could be retrieved from the backend and displayed accurately in the frontend.

Article Updating: Confirming that existing articles could be updated with new information or modifications.

Article Deletion: Validating the deletion process, ensuring that articles could be removed from the database without any errors.



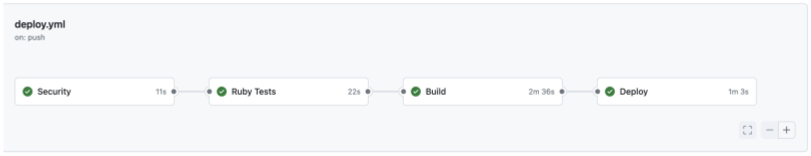
```
e2e
    addArticle.cy.ts
    articleDetails.cy.ts
    articleList.cy.ts
    editArticle.cy.ts
```



```ruby
require 'test_helper'

class ArticlesControllerTest < ActionDispatch::IntegrationTest
  setup do
    @article = articles(:one)
  end

  test 'should get index' do
    get api_v1_articles_url, as: :json
    assert_response :success
  end

  test 'should create article' do
    assert_difference('Article.count') do
      post api_v1_articles_url,
           params: { article: { body: @article.body, published: @article.published, title: @article.title } }, as: :json
    end

    assert_response :created
  end

  test 'should show article' do      You, last week • first commit: backend 'finished' and react proj…
    get api_v1_article_url(@article), as: :json
    assert_response :success
  end
```

Unit Testing vs. Integration Testing: Unit testing ensures individual components function correctly, catching bugs early and maintaining code quality; they are usually . Integration testing validates interactions between components, ensuring seamless system workflows and reliability in production environments. Both are integral to software development, improving quality, reliability, and maintainability.

Deployment and Cloud Integration:



I used Docker to containerize app. With Docker Compose, I orchestrated the deployment of multiple containers as a unified application stack. A CD/CI pipeline using GitHub Actions automated the build, test, and deployment processes, ensuring easy integration with the AWS cloud platform. Finally, I provisioned an AWS EC2 instance to host the "Dockerised" application, leveraging its scalability, reliability, and ease of management. It's important to note that this app does not store data between iterations, since it's a demo app. This was purposely done this way to reduce complexity.

Links:

Rails/Backend: http://34.248.117.130:3000/

React App: http://34.248.117.130:3001/

HTML Client: http://34.248.117.130:8000/

Repository: https://github.com/lmartins18/cloud-application-development-ca/